- Assignment 2, Midterm grades this week
- Course Projects: round 2 meetings next Friday
- Next Tuesday: Guest speaker for first part
WHAT WE KNOW SO FAR
CONTINUOUS OPERATOR MODEL

- Long-lived operators
- Mutable State
- Distributed Checkpoints
- High overhead for Fault Recover
- Stragglers?

Diagram:
- Driver
- Control Message
- Task
- Network Transfer

Software tools:
- Flink
- Naiad
GOALS

1. Scalability to hundreds of nodes

2. Minimal cost beyond base processing (no replication)

3. Second-scale latency

4. Second-scale recovery from faults and stragglers
DISCRETIZED STREAMS
DISCRETIZED STREAMS (DSTREAMS)

Approach
- Use *short, stateless, deterministic tasks*
- Store *state* across tasks as in-memory RDDs
- Fine-grained tasks $\rightarrow$ Parallel recovery / speculation

Model
- Chunk inputs into a number of *micro-batches*
- Processed via parallel operations (i.e., map, reduce, groupBy etc.)
- Save intermediate state as RDD / write output to external systems
COMPUTATION MODEL: MICRO-BATCHES

The diagram illustrates the micro-batch computation model. It includes:

- **Driver** (red): Initiates the process.
- **Task** (green): Processing units.
- **Control Message** (arrow): Direction of data flow.
- **Network Transfer** (blue): Data transfer between tasks.

The diagram shows a sequence of tasks connected by control messages and network transfers, indicating the flow of computation.
pageViews = readStream(http://..., "1s")

ones = pageViews.map(
    event => (event.url, 1))

counts = ones.runningReduce(
    (a, b) => a + b)

EXAMPLE

\[
\text{pageViews} = \text{readStream}(\text{http://...}, "1s")
\]

\[
\text{ones} = \text{pageViews.map}(\text{event} \Rightarrow (\text{event.url}, 1))
\]

\[
\text{counts} = \text{ones.runningReduce}(a, b) \Rightarrow a + b
\]
DSTREAM API

Output operations
   save output to external database / filesystem

Transformations
   Stateless: map, reduce, groupBy, join
   Stateful:
      window(“5s”) \(\rightarrow\) RDDs with data in \([0,5), [1,6), [2,7)\)
      reduceByWindow(“5s”, (a, b) \(\Rightarrow\) a + b) \(\rightarrow\) incremental aggregation
ASSOCIATIVE, INVERTIBLE

Add previous 5 each time

(a) Associative only

(b) Associative & invertible
OTHER ASPECTS

Tracking State: streams of (Key, Event) $\rightarrow$ (Key, State)
- Initialize: Create a State from the first event
- Update: Return new State given, old state and event
- Timeout for dropping old states.

Unifying batch and stream
- Join DStream with static RDD
- Attach console and query existing RDDs
- Shared codebase, functions etc.

```scala
events.track(
  (key, ev) => 1,
  (key, st, ev) =>
    ev == Exit ?
    null : 1,
  "30s")
```
SYSTEM IMPLEMENTATION

Master
- D-Stream lineage
- Input tracker

Worker
- Input receiver
- Task execution
- Block manager
- Comm. Manager

Worker
- Input receiver
- Task execution
- Block manager
- Comm. Manager

Client

 replication of input & checkpointed RDDs

New

Modified
OPTIMIZATIONS

Network Communication
   Rewrote Spark’s data plane to use asynchronous I/O

Timestep Pipelining
   No barrier across timesteps unless needed
   Tasks from the next timestep scheduled before current finishes

Checkpointing
   Async I/O, as RDDs are immutable
   Forget lineage after checkpoint
Worker failure
  - Need to recompute state RDDs stored on worker
  - Re-execute tasks running on the worker

Strategy
  - Run all independent recovery tasks in parallel
  - Parallelism from partitions in timestep and across timesteps
```scala
pageViews = readStream(http://..., "1s")

ones = pageViews.map(
    event => (event.url, 1))

counts = ones.runningReduce(
    (a, b) => a + b)
```
FAULT TOLERANCE

Straggler Mitigation

Use speculative execution
Task runs more than 1.4x longer than median task → straggler

Master Recovery

- At each timestep, write out graph of DStreams and Scala function objects
- Workers connect to a new master and report their RDD partitions
- Note: No problem if a given RDD is computed twice (determinism).
Expressiveness
- Current API requires users to “think” in micro-batches

Setting batch interval
- Manual tuning. Higher batch $\rightarrow$ better throughput but worse latency

Memory usage
- LRU cache stores state RDDs in memory
SUMMARY

Micro-batches: New approach to stream processing

Higher latency for fault tolerance, straggler mitigation

Unifying batch, streaming analytics